Trait-oriented Programming in Java 8

Viviana Bono, Enrico Mensa, Marco Naddeo

Trait-oriented Programming in Java 8 *

Viviana Bono Enrico Mensa Marco Naddeo
Dipartimento di Informatica, University of Torino, Italy
{bono,naddeo}@di.unito.it, enrico.mensa@gmail.com

Abstract

Java 8 was released recently. Along with lambda expressions, a new language construct is introduced: default methods in interfaces. The intent of this feature is to allow interfaces to be extended over time preserving backward compatibility. In this paper, we show a possible, different use of interfaces with default methods: we introduce a trait-oriented programming style based on an interface-as-trait idea, with the aim of improving code modularity. Starting from the most common operators on traits, we introduce some programming patterns mimicking such operators and discuss this approach.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features

Keywords Java 8, default method, trait, programming pattern, code modularity.

1. Introduction

From the point of view of the language constructs, the most prominent addition in Java 8 is the *lambda-expression* construct, that comes along with an apparently secondary construct, that is, the *default method* (aka *virtual extension method*, aka *defender method*) in interfaces. The primary intent of this feature is to allow interfaces to be extended over time preserving backward compatibility. These features of Java 8 are described in the proposal *JEP 126* (JDK Enhancement Proposal 126) *Lambda Expressions & Virtual Extensions Methods* [9]. JEP 126 is a follower of the Project Lambda, that corresponds to JSR 335 (Java Specification Request 335) [17].

A *default method* is a virtual method that specifies a concrete implementation within an interface: if any class implementing the interface will override the method, the more specific implementation will be executed. But if the default method is not overridden, then the default implementation in the interface will be executed. An already paradigmatic example of use of default methods to preserve backward compatibility concerns the interface `Collection<T>`, already present in previous releases of Java. Thanks to lambda expressions, now it is possible to introduce a `forEach(LE)` method that takes a lambda expression `LE` as an argument, coding a function to be applied to all of the elements in the collection. However, adding this method as abstract to `Collection<T>` interface would mean breaking backward compatibility: in other words, every old class implementing `Collection<T>` interface would have to change by adding an implementation of `forEach(LE)`, unless a default method is present in the interface. Moreover, default methods may help avoiding code duplication.

In previous Java releases, interfaces were to provide multiple type inheritance, in contrast to the class-based single implementation inheritance. Java 8 interfaces, instead, introduce a form of multiple implementation inheritance, too. Therefore, they are similar to *traits* [10], which are sets of (required and supplied) methods. [1] Java 8 interfaces, then, can be exploited to introduce a *trait-oriented* programming style. Note that we are not proposing a linguistic extension of Java 8 with traits, but programming patterns within Java 8, with the goal of improving *code modularity* and, therefore, *code reuse*. Starting from operators on traits [10], we introduce some Java 8 programming patterns mimicking such operators and discuss this approach.

The paper is organized as follows: Section 2 illustrates briefly the *trait* construct, Section 3 introduces Java 8 *default methods*, Section 4 proposes the programming patterns inspired by the trait operators, Section 5 shows an example of use, Section 6 makes some comparisons with related work, and Section 7 draws some conclusions.

2. What are traits?

The possibility of *composition* and *decomposition* of code are important characteristics to care about in a programming language. Let us point out some problems of (single and multiple) inheritance concerning composability:

- *Duplicated features*. Single inheritance is the basic form of inheritance; thanks to that we can reuse a whole class (and also add some features). But sometimes we want to express something that is much too complex to be implemented with single inheritance. For example, we could have a class `Swimming` (that gives features for swimming animals) and a class `Flying` (that gives features for flying animals). What if we want to create an animal that can both swim and fly, as swans? We can inherit only from `Swimming` or `Flying` but not both, so we will have to duplicate some of the existing features in the `Swan` class.
- *Inappropriate hierarchies*. Instead of duplicating methods in the lower classes, we can bring those methods up in the hierarchy; however, this way we violate the semantics of the upper classes.
- *Conflicting features*. If we have multiple inheritance (as C++ does) a common problem is how to treat conflicts. Method conflicts can be solved (for example, thanks to overriding), but

* This material is based upon work supported by MIUR PRIN Project CINA Prot. 2010LHT4KM and Ateneo/CSP Project SALT.

[1] This is pointed out in many places, see, for instance, [18].

conflicting attributes are more problematic. It is never clear if a conflicting attribute should be inherited once or twice and how these attributes should be initialized.

Traits are a possible solution to these problems. A trait is a "*simple conceptual model for structuring object-oriented programs*" [10] and it is a collection of methods. This is very important: these traits are statless, they cointain only methods, therefore every conflict of state is avoided. Only method name conflicts must be dealt with, explicitly, by the programmer.

Every trait can define *required methods* and *required fields*. In the considered model, required fields are indirectly modelled via required setter and getter methods. A trait can be defined directly (by specifying its methods) or by composing one or more traits. The composition is performed by means of the following operators:

- *Symmetric Sum*: a new trait is defined by combining two or more existing traits whose method sets are disjoint. In the case the sets are not disjoint, conflicts arise.
- *Trait overriding*: a new trait is defined by adding method(s) to an existing trait. If an already present method is added, the old version is overridden.
- *Exclusion*: a new trait is defined by excluding a method from an existing trait.
- *Aliasing*: a new trait is defined by adding a second name to a method from an existing trait. This is useful if the original name was excluded after resolving a conflict. Note that, if a recursive method is aliased, the recursive call will be done on the original method.

These operators are from the original proposal [10]. Other operators were introduced in further works (a comprehensive list of operators with relations among them can be found in [5]). We focus on this particular set as we think they are the most interesting ones from a programming point of view.

The original definition of traits says that trait and class usages are separated: the first ones are units of reuse, while the second ones are generator of instances. A class can be specified by composing a superclass with a set of traits and some *glue methods* (aka *glue code*). Glue methods are written inside a class and make it possibile the connection between different traits. An example of glue code are the setter/getter methods, that allow methods in traits to access the fields.

Trait composition respects the following three rules [10]:

- Methods defined in a class itself take precedence over methods provided by a trait. This allows glue methods defined in the class to override methods with the same name provided by the traits.
- Flattening property: a non-overridden method in a trait has the same semantics as if it were implemented directly in the class.
- Composition order is irrelevant. All the traits have the same precedence, and hence conflicting trait methods must be explicitly disambiguated.

Method name conflicts can be resolved by introducing appropriate glue methods in classes which redefine the conflicting methods, or thanks to two operators:

- with trait override, by adding one method with the same name, which hides the previous implementations and may call whichever of them;
- with exclusion, by excluding all but one of the conflicting methods.

3. On default methods

The role of an interface up to Java 7 was to give a contract to the user (that is, a type), but not to specify any detail of the contract itself (that is, the implementation). The main characteristic of default methods (introduced by a keyword `default`) is that they are virtual like all methods in Java, but they provide a default implementation within an interface.

Java 8 method resolution is defined in [12] and its formalization in a Featherweight-Java style [14] can be found in [13]. To summarize it, we take the four (informal) rules about method linkage from [13]:

- A method defined in a type takes precedence over methods defined in its supertypes.
- A method declaration (concrete or abstract) inherited from a superclass takes precedence over a default inherited from an interface.
- More specific default-providing interfaces take precedence over less specific ones.
- If we are to link `m()` to a default method from an interface, there must be a unique most specific default-providing interface to link to, otherwise the compiler signals a conflict.

From these dispatch rules, we can extrapolate some examples of behaviour that can help the reader to understand the default method construct.

A first example. If the class that implements the interface using default methods does not override those methods, the default implementation provided in the interface will be executed.

```
interface A {
  default void m()
    {out.println("Hi, I'm interface A");}
}
class B implements A {}
//doesn't override m

public class FirstDM {
  public static void main(String[] args) {
    B b = new B();
    b.m();
  }
}
```

The output will be: `Hello from interface A`.

Classes always win. Here there is an example:

```
interface A {
  default void m()
    {out.println("Hi, I'm interface A");}
}
class B implements A {
  //overrides m
  public void m()
    {out.println("Hi, I'm class B");}
}
public class SecondDM {
  public static void main(String[] args) {
    B b = new B();
    b.m();
  }
}
```

The output will be: `Hi, I'm class B`.

The most specific interface wins. If no class overrides a default method, the default method with the most specific implementation will be executed:

```
interface A {
```

```
  default void m()
    {out.println("Hi I'm interface A");}
}
interface B extends A {
  default void m()
    {out.println("Hi I'm interface B");}
} //more specific because of the 'extends'

class C implements A, B { }

public class ThirdDM {
  public static void main(String[] args) {
    C c = new C();
    c.m();
  }
}
```

The output will be: `Hi, I'm interface B`.

Conflicts are not always avoidable. If a unique most specific default-providing interface is not found, an error will occur:

```
interface A {
  default void m()
    {out.println("Hi I'm interface A");}
}
interface B {
  default void m()
    {out.println("Hi I'm interface B");}
}

class C implements A, B { }

public class FourthDM {
  public static void main(String[] args) {
    C c = new C();
    c.m();
  }
}
```

The compiler says: `class C inherits unrelated defaults for m() from types A and B - class C implements A, B`.

How to resolve conflicts. The construct `X.super.m()` can be used, where `X` is one of the direct superinterfaces containing the default method `m()`:

```
interface A {
  default void m()
    {out.println("Hi I'm interface A");}
}
interface B {
  default void m()
    {out.println("Hi I'm interface B");}
}
class C implements A, B {
  //calls m in A
  public void m()
    {A.super.m();}
}
public class FifthDM {
  public static void main(String[] args) {
    C c = new C();
    c.m();
  }
}
```

The output will be: `Hi I'm interface A`.

Note that this new construct is just for resolving conflicts while using default methods and not for a general purpose [12].

About abstract methods. We said that classes always win over interfaces. This is true also when classes are abstract:

```
interface A {
  default void m()
    {out.println("Hi I'm interface A");}
}
abstract class B {
  abstract void m();
}

class C extends B implements A { }

public class SixthDM {
  public static void main(String[] args) {
    C c = new C();
    c.m();
  }
}
```

The compiler says: `C is not abstract and does not override abstract method m() in B - class C extends B implements A`. This happens because the abstract declaration of `m()` in `B` takes precedence over the default declaration in `A`.

4. A guide to trait-oriented programming

Java 8 interfaces play the role of traits, with default methods as provided methods and abstract methods as required methods. We will refer to an interface with this role with the term "trait" and we introduce the convention that such an interface will be named with a name starting with `T` or `Trait`. As within stateless traits, required fields are encoded as required accessory (getter and setter) methods, that is, as abstract methods, whose implementation will be provided as glue code by the class implementing the traits.

In order to introduce the trait-oriented programming style, we propose some programming patterns to match the trait operators listed in Section 2.

Symmetric sum. This provides the fundamental feature of multiple inheritance. With "symmetric" it is meant that all the addends of a sum are peers, implying that, in the case of a conflict, it is up to the developer to deal with it. The first example shows a case of a sum without conflicts. We have three traits: `TMouth`, `TEyes` and `TTail`:

```
public interface TMouth {
  default void makeASound()
    {out.println("Yaaaawn");}
  default void eat(String s)
    {out.println("I'm eating "+s);}
}


public interface TEyes {
  default void lookAround()
    {out.println("I'm looking");}
  default void blink()
    {out.println("I'm blinking");}
}


public interface TTail {
  default void shakeTail() {
    out.println("Wuush, I'm shaking my tail.");
  }
}
```

Then a new trait, `TCat`, puts together all the features previously defined, and a class implements it:

```
public interface TCat
    extends TEyes, TMouth, TTail {
  default public void purr()
    {out.println("PuUurRrRr");}
}
```

```
public class MyCat implements TCat {
  private String name;
  public MyCat(String n)
    {this.name = n;}

  public static void main(String[] args) {
    MyCat jacky = new MyCat("Jacky");
    jacky.eat("Meat");
  }
}
```

The output will be: `I'm eating Meat"`.

Trait overriding. The override operator defines a new trait by adding one or more methods to an existing trait:

```
public interface TraitA {
  default void m()
    {out.println("I am m in TraitA");}
}

public interface TraitB extends TraitA {
  /** overrides TraitA, adding
      a new feature **/
  default void m2()
    {out.println("I am m2 in TraitB");}
}

public class C implements TraitB {
  public static void main(String[] args) {
    C c = new C();
    c.m();
    c.m2();
  }
}
```

Both methods `m` and `m2` are callable, therefore the output will be: `I am m in TraitA`, `I am m2 in TraitB`.

Trait overriding can be used to solve conflicts. If we add a method `close()` in both traits `TMouth` and `TEyes` (introduced above), we get from the compiler: `MyCat.java:1: error: class MyCat inherits unrelated defaults for close() from types TEyes and TMouth public class MyCat implements TCat`.

Notice that in the overriding version of `close()`, we use the construct `X.super.m()`:

```
public interface TCat
    extends TEyes, TMouth, TTail {

  /** Conflict resolution **/
  default void close()
    {TEyes.super.close();}

  default public void purr()
    {out.println("PuUurRrRr");}
}
```

The method `close()` that will be executed is the one from the `TEyes` trait. The `close()` method from the `TMouth` trait is not lost, as it can be aliased. However, notice that the use of the `X.super.m()` feature reduces the low coupling between `TCat` and `TEyes`: if some day the `close()` in `TEyes` method will change (for example by adding a parameter to it), also the `close()` method inside `TCat` must change.

Exclusion. Exclude is a tough operator. In [12], it was described the possibility to remove a default method by using the `default none` keyword, but this has not made its way in the official Java 8 release (dealing with negative information is never easy). A proposal for an exclude programming pattern, then, can use a well-know workaround, i.e., we can exclude a method by redefining it with an empty body or by throwing an exception. We prefer the second alternative. Consider this trait:

```
public interface TraitA {
  default void m()
    {out.println("I am m in TraitA");}
  default void q()
    {out.println("I am q in TraitA");}
}
```

If we want to exclude `m()`, we can do as follows:

```
public interface TraitB extends TraitA {
  default void m() {
    String s = "Method not understood";
    throw new UnsupportedOperationException(s);
  }
}

public class C implements TraitB {
  public static void main(String[] args){
    C c = new C();
    c.m();
    c.q();
  }
}
```

The first method call throws the exception. The second one would print `I am q in TraitA`.

Note that this programming pattern works well with respect to symmetric sum: if in all summed traits we have a method that we want to exclude, then this pattern will exclude simultaneously all upper method versions. However, we do not exclude the method for real, we just make unavoidable the upper implementation by overriding it, therefore Java introspection can still detect the excluded method: `C.class.getMethod('m')` still gets an answer.

Notice, however, that it is not possible to call in a new trait the excluded `TraitA` version of the method `m`:

```
public interface TraitB1 extends TraitB {
  /** It tries to rehabilitate the version
      from TraitA, excluded by TraitB **/
  default void m() {
    TraitA.super.m(); //does not compile
  }
}
```

If we try to compile the above code, we obtain an error: `not an enclosing class: TraitA`.

Aliasing. The alias operator provides another, alternative, name for referring to a certain method. Consider this trait:

```
public interface TraitA {
  default void mOneA()
    {out.println("I'm mOneA in A");}
  default void mTwoA()
    {out.println("I'm mTwoA in A");}
}
```

Now, in a new trait, we create an alias for the `mTwoA()` and we test it:

```
public interface TraitB extends TraitA {
  /** Aliasing mTwoA() in
      aliasMTwoA() **/
  default void aliasMTwoA()
    {mTwoA();}
}

public class MyB implements TraitB {
  public static void main(String[] args) {
    MyB mc = new MyB();
    mc.aliasMTwoA();
    mc.mTwoA();
  }
}
```

The output will be: `I'm mTwoA in A`, twice.

When applying the alias programming pattern, attention must be paid to the alias name, as it is possible to override by mistake another method of the upper trait.

4.1 On the return type of methods

In Java, the name of a method is bound forever to its first introduction in terms of the return type. This makes the reuse of the name fragile, if we want to change the return type. We discuss this issue by means of an example. We want to develop a stack data structure (this example is taken from [5]). First of all, we show a single inheritance version:

```
public interface IStack {
  /* Tells if the stack is empty */
  public boolean isEmpty();
  /* Adds one item on the stack */
  public void push(Object obj);
  /* Removes and returns the first
     object on the stack */
  public Object pop();
}


public class Stack implements IStack {
  List<Object> l;

  public Stack()
    { l = new LinkedList<Object>(); }
  public boolean isEmpty()
    { return l.isEmpty(); }
  public void push(Object obj)
    { l.add(obj); }
  public Object pop() {
    if (!isEmpty())
      return l.remove(l.size()-1);
    else
      return null;
  }
}
```

Now, suppose that we want to implement another stack, with this interface:

```
public interface IStackAlt {
  public boolean isEmpty();
  public void push(Object obj);
  /* Removes the first object on the stack */
  public void pop();
  /* Returns the first object on the stack
    (without removing it) */
  public Object getTop();
}
```

As we can see, this interface is different from `IStack` because of two methods: `pop()` is now `void`, and we have an additional method `getTop()`. We can implement this interface as follows:

```
public class StackAlt implements IStackAlt {
  List<Object> l;

  public StackAlt()
    { l = new LinkedList<Object>(); }
  public boolean isEmpty()
    { return l.isEmpty(); }
  public void push(Object obj)
    { l.add(obj); }
  public void pop()
    { if (!isEmpty()) l.remove(l.size()-1); }
  public Object getTop()
    { if (!isEmpty()) return l.get(l.size()-1);
      else return null; }
}
```

Notice that both methods `isEmpty()` and `push()` were already implemented inside the `Stack` class and we had to re-implement them inside the `StackAlt` class.

We exploit now our trait-oriented approach to promote code reuse and backward compatibilty when possible, while pointing out the problems related to override.

From a non-void method to a void one. First of all, we introduce a `TStack` trait that defines all the operations:

```
public interface TStack {
  public List<Object> getStructure();

  default boolean isEmpty()
    { return getStructure().isEmpty(); }
  default void push(Object obj)
    { getStructure().add(obj); }
  default Object pop() {
    if (!isEmpty()) {
      int pos = getStructure.size()-1;
      Object o = getStructure().get(pos);
      getStructure().remove(pos);
      return o;
    }
    return null;
  }
}
```

Notice the abstract method `getStructure()`: it is a getter method to access the stack structure, that will be implemented as a field in a class, together with this method. The implementation of `TStack` is as follows:

```
public class Stack implements TStack {
  List<Object> l;
  public Stack()
    { l = new LinkedList<Object>(); }
  /* Glue Code */
  public List<Object> getStructure()
    { return l; }
}
```

Note we put some glue code to provide the previously mentioned `getStructure()` method.

Now, we want to introduce a new method `getTop()` and we want to change the old `pop()` that was returning an `Object` into a `void` version. The first goal is easy, we can use the trait override pattern, however we encounter some problems with the `pop()` method:

```
public interface TStackAlt extends TStack {
  /** We redefine pop simulating
      the void return type **/
  default Object pop() {
    if (!isEmpty()) {
      int pos = getStructure.size()-1;
      getStructure().remove(pos);
    }
    return null;
  }

  /** If we could, we would have done:
  default Object pop() {
    String s = "Message not understood";
    throw new UnsupportedOperationException(s);
  }

  default void pop() {
    int pos = getStructure.size()-1;
    getStructure().remove(pos);
  }
  but in Java we cannot have two methods
  with same name and number of parameters
  which differ only for their return types. **/
```

```
  /** We make the old pop still available
      (optional) **/
  default Object popTop() {
    return TStack.super.pop();
  }
  /** Trait Override **/
  default Object getTop() {
    if (!isEmpty()) {
      int pos = getStructure.size()-1;
      return getStructure().get(pos);
    }
    return null;
  }
}
```

In Java we cannot have two methods with same name and number of parameters which differ only for their return types. We did, in fact, provide an ad-hoc solution, by returning `null` in the new version of `pop()`. This is an implementing class:

```
public class StackAlt implements TStackAlt {
  List<Object> l;
  public StackAlt()
    { l = new LinkedList<Object>(); }
  /* Glue Code */
  public List<Object> getStructure()
    { return l; }
}
```

Notice that this solution preserves backward compatibility and it can be applied in similar cases. With respect to the single-inheritance version, the methods `isEmpty()` and `push()` are not duplicated anymore, the class tree is clearer, we provided a new `pop()` method but we also made the old one still accessible.

From a void method to a non-void one. Now `TStack` defines a `void pop()` and `TStackAlt` defines a new `Object`-returning version of it. We have the two traits defined as follows:

```
public interface TStack {
  public List<Object> getStructure();
  default boolean isEmpty()
    { return getStructure().isEmpty(); }
  default void push(Object obj)
    { getStructure().add(obj); }
  default void pop() {
    if (!isEmpty()) {
      int pos = getStructure.size()-1;
      getStructure().remove(pos);
    }
  }
  default Object getTop() {
    if (!isEmpty()) {
      int pos = getStructure.size()-1;
      return getStructure().get(pos);
    }
    return null;
  }
}


public interface TStackAlt extends TStack {
  /** We cannot change the return type
      of the pop() method from void to
      Object. So we make an alternative
      pop, using another name.
  **/
  default Object altPop() {
    Object o = getTop();
    pop();
    return o;
  }
}
```

This case is similar to the previous one, however no workaround is possible. All we could do was to define another "pop" method that returns an `Object` (method `altPop()`).

The implementing class `StackAlt` is the same as in the previous case. Notice, however, that any call to `pop()` in a client will have to be changed into a call to `altPop()`: backward compatibility is broken.

The covariant override. In Java the override behaves *covariantly*: an overriding method may have a return type that is a subtype of the method it overrides. Therefore, we may do something like this:

```
public interface TraitA {
  default Object m()
    { /* something */ }
}
public interface TraitB extends TraitA {
  default String m()
    { /* something else */ }
}
```

This works because `String` is a subtype of `Object`. Of course we cannot do the opposite, that is, change the return type from `String` to `Object`, but a case of trait manipulation requiring such a change of signature would probably make little sense.

5. An example

We present now a small case study: the skeleton implementation of a variant of a classic game (often used when presenting alternative reuse mechanisms to inheritance and design patterns). The Java 8 code is presented in Appendix A and the corresponding diagram is shown in Figure **??**.

There is a character with the goal of collecting as many coins as possible while she is moving from one place to another. Sometimes, the character enters a room with three doors to be chosen among. A door can be opened only if it is unlocked, each door has its own features and provides a certain amount of coins. In particular, we develop a room with three different doors. This example is suited to show code reuse as any kind of door can appear in different rooms and any feature of the doors can be exploited on elements that are not doors, in different situations during the game.

We start with a base door. If the door is not locked, it is possible to open it. Once opened, the door gives some coins (a negative value corresponds to "door still locked"). As an additional action, it is possible to knock the door and try to get some other coins. Now, we develop three features (we can, however, imagine a lot of them), that can be applied to the doors but also to other elements of the game. These are a coin counter, a chest of coins, an enchantment that can give or take coins, and they are represented with three traits, `TCounter`, `TChest`, and `TEnchanment`. In the traits there are some required methods such as `getEnchantMaxCoins()` and `getDoorMaxCoins()`: it is up to the developer of an element to decide the maximum amount of coins that it can give.

We can compose our `TDoor` with different features and then create a class implementing each door (and providing the required glue code): for example, an enchanted door, represented by a class `EnchantedDoor` that uses a trait `TEnchandedDoor`. Note that the class `EnchantedDoor` provides the required glue code by implementing `getEnchantMaxCoins()` and `getDoorMaxCoins()`.

Similarly, we can implement a trait `TChestedDoor` and a class `ChestedDoor`. We solve a conflict between the two `open()`'s methods (one from `TDoor` and the other from `TChest`). First, we alias the `open()` from the `TChest` trait as `openChest()`. Then, we use the trait overriding to reuse the conflicting name and re-implement the `TDoor`'s `open()`.

We can also build a door that may give coins when knocked a certain number of times (`TKnockDoor` and a class `KnockDoor`).

Now we can create a room with three doors (an alternative implementation could use a `TRoom` trait that provides general methods for a room and then a class `DoorsRoom` that implements it). Finally we put everything together. We create: a `Game` class that references the rooms of the game and the player; a `Player` class (again, we could introduce a `TPlayer` trait representing a basic player, that could be combined with other traits to obtain different categories of players such as premium, demo, etc).

Every trait is reusable inside the project and every feature is stand-alone. However, there is a drawback: some of glue code inside the implementing classes is duplicated and, even if field accessory methods might be generated authomatically, this is detrimental for the code reuse (even though it seems a good idea that the developer of a class can decide how to access its variables). Fortunately, if trait composition is used in synergy with class inherintance, reuse can be improved. In our case, it is enough to regroup all the replicated glue code (and the relative fields) into a class `CommonDoor` (see Figure **??**).

6. Related work

Traits as in [10] have been fully implemented in Smalltalk-Pharo [19]. A form of traits is present in PHP 5.4 [20], and in the literature there are a few other different models and implementations: for instance, in [5] and [22] there are two proposals for traits in a Java-like language, and [3] presents a version of traits with state (however, at the best of our knowledge, no satisfactory versions of stateful traits have been proposed so far).

Traits and *mixins* are related. Both constructs exploits composition instead of inheritance as a mechanism for software reuse and they are alternatives to multiple inherintance. Mixins [1, 4, 6, 7, 11, 24] are essentially subclasses parametric over their superclass and can be seen as a form of linearized multiple inheritance. We outline the three main differences between traits and mixins:

- Mixins are stateful, as they can define fields (providing reuse of behaviour *and* state), while traditional traits do not.
- Mixins use *implicit* conflict resolution (the resolution is based on the semantics of the language, for instance, it depends on the relative order of the components in a composition), while traits use *explicit* conflict resolution via the application of operators (the responsability of solving the conflicts is of the developer).
- Mixins depend on linearization, traits are flattened [10].

There are libraries for modifying classes via bytecode manipulation [15] and to extend classes and implement interfaces at runtime [8], performing then a sort of runtime code composition. We believe that trait/mixin composition can be used sometimes to solve problems similar to those solved by this kind of runtime composition; still, this is far from trait/mixin composition, as, first of all, it is not type-checked. Moreover, if traits and mixins are instruments for software modularization and reuse, these libraries are meant to solve different problems, such as run-time code debugging.

Aspect-oriented programming [2, 16] shares with traits and mixins the goal of software reuse, moreover aspects, traits and mixins can be all statically typed. However, its applications differ, as traits and mixins have the goal of organizing code, while aspects contain those parts of code that are cross-cutting concerns. If traits and mixins can have a more general application, the code composition based on aspects is more fine-grained, as it is perfomed at the level of methods (via pointcut definitions) and not at the level of the containers of the methods.

In [21] there are two proposals to model a mixin-based programming style in Java 8, that is, a stateful approach. The first one exploits lambda expressions to model the state but suffers from some problems related to the runtime semantics of lambda expressions. The second (successful) proposal relies on the *virtual field pattern*, which is nothing else than the trait glue-code technique (i.e., a field is requested by defining one or more required accessory methods in the trait, that will be implemented in the class) that we also exploit. However, this proposal does not consider the trait operators in detail.

Another approach for a trait-oriented programming style in Java 8 is from [23]. Here conflicts are solved by applying an instance of the Decorator pattern. However, with this solution, no new features of Java 8 are used. Still, this would be a solution in Scala [24], as with Scala traits (which are, indeed, mixins) it is straighforward to program such a pattern.

7. Conclusions

Traits (and mixins) help the developer in thinking the elements of a project as stand-alone units of reuse that can be composed together, changed and remodelled. This paper does not introduce a trait-based language, but offers a view on how default methods can be exploited to promote and improve code modularization via an interface-as-trait programming approach. To this aim, Java 8 interfaces play, then, the role of traits, where abstract methods are the *required* methods (including the field accessory methods), and default methods are the *provided* methods. We have described some programming patterns inspired by the trait operators present in [10]: symmetric sum (to form a new trait by composing two or more existing traits), trait overriding (to form a new trait by adding methods to an existing trait), exclusion (to form a new trait by deleting a method from an existing trait), aliasing (to form a new trait to give a method an alternative name). The symmetric sum might introduce conflicts among method names, that must be solved by the programmer with the use of trait overriding and exclusion. Our programming patterns rely heavily on Java override, therefore some attention must be paid when there is not only a change of name, but also a change of signature (being in a typed setting). In Section 4.1, we described the possible problems that can arise and hinted some solutions.

As our interfaces-as-traits are stateless and accessory methods are the only (indirect) way to specify fileds in trait, our approach imposes a restriction on visibility of fields. However, this is exactly how it works within stateless traits [10].

As it is usual with traits (and mixins), it seems that our approach has an impact on code modularity (see Section 5), which implies more reusability and maintainability. It would be also interesting to refactor a large-scale, real-world example by applying our programming patterns and then use appropriate metrics (e.g., LOC) to measure the before- and after-factorization performances.

At the best of our knowledge, our proposal is the first one to explore the possibilty of a trait-oriented programming style in Java 8. We have shown that it is possible to implement the standard operators on traits with little work-around. A formal mapping from traits to Java 8 may be achieved by using Featherweight Java-style calculi [14].

Another possible direction to explore is the possibility of excluding default methods (starting from [12], where it was described a `default none` keyword). Moreover, we believe that our work could be also the base for reflecting about which form of traits (or even mixins) might be good to be added as a language construct in future releases of Java.

References

[1] D. Ancona, G. Lagorio, and E. Zucca. Jam — a smooth extension of Java with mixins. In *Proc. ECOOP '00*, volume 1850 of *LNCS*, pages 145–178. Springer-Verlag, 2000.

[2] AspectJ Documentation. http://www.eclipse.org/aspectj/docs.php.

[3] A. Bergel, S. Ducasse, O. Nierstrasz, and R. Wuyts. Stateful traits and their formalization. *Computer Languages, Systems and Structures*, 34(2-3):83–108, 2008. ISSN 1477-8424. http://dx.doi.org/10.1016/j.cl.2007.05.003.

[4] V. Bono, A. Patel, and V. Shmatikov. A Core Calculus of Classes and Mixins. In *Proc. ECOOP '99*, volume 1628 of *LNCS*, pages 43–66. Springer-Verlag, 1999.

[5] V. Bono, F. Damiani, and E. Giachino. On traits and types in a Java-like setting. In G. Ausiello, J. Karhumki, G. Mauri, and C.-H. L. Ong, editors, *IFIP TCS*, volume 273 of *IFIP*, pages 367–382. Springer, 2008. ISBN 978-0-387-09679-7.

[6] V. Bono, J. Kusmierek, and M. Mulatero. Magda: A new language for modularity. In *ECOOP*, pages 560–588, 2012.

[7] G. Bracha. *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance*. PhD thesis, The University of Utah, 1992.

[8] Code Generation Library. http://cglib.sourceforge.net/.

[9] J. D. Darcy. JEP 126: Lambda Expressions & Virtual Extension Methods. http://openjdk.java.net/jeps/126.

[10] S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts, and A. Black. Traits: A mechanism for fine-grained reuse. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 28, no. 2:331–388, 2006.

[11] M. Flatt, S. Krishnamurthi, and M. Felleisen. A programmer's reduction semantics for classes and mixins. In *Formal Syntax and Semantics of Java*, pages 241–269, London, UK, UK, 1999. Springer-Verlag. ISBN 3-540-66158-1. http://dl.acm.org/citation.cfm?id=645580.658808.

[12] B. Goetz. Interface evolution via virtual extensions methods. http://cr.openjdk.java.net/∼briangoetz/lambda/Defender%20Methods%20v4.pdf, June 2011.

[13] B. Goetz and R. Field. Featherweight Defenders: A formal model for virtual extension methods in Java. http://cr.openjdk.java.net/∼briangoetz/lambda/featherweight-defenders.pdf, March 2012.

[14] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *TOPLAS*, 23(3):396–450, 2001.

[15] Javassist. http://www.csg.ci.i.u-tokyo.ac.jp/∼chiba/javassist/.

[16] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. Getting started with AspectJ. *Commun. ACM*, 44(10):59–65, 2001. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/383845.383858.

[17] Lambda Expressions. Lambda Expressions for the Java Programming Language. http://openjdk.java.net/projects/lambda/.

[18] A. C. Oliver. Love and hate for Java 8. http://m.javaworld.com/javaworld/jw-07-2013/130725-love-and-hate-for-java-8.html?mm_ref=https://www.google.it.

[19] Pharo. http://www.pharo-project.org/home.

[20] PHP 5.4.0 Release Announcement. http://php.net/releases/5_4_0.php.

[21] F. Sarradin. Java 8: Now you have mixins? http://kerflyn.wordpress.com/2012/07/09/java-8-now-you-have-mixins/.

[22] C. Smith and S. Drossopoulou. Chai: Traits for Java-like Languages. In *Proc. ECOOP '05*, volume 3586 of *LNCS*, pages 453–478. Springer-Verlag, 2005.

[23] R. Sollid. Java 8 and mixin with default methods. http://reidarsollid.com/2013/03/28/java-8-and-mixin-with-default-methods/, March 2013.

[24] The Scala Group. Scala Website. http://www.scala-lang.org/.

A. The example's code

```
import java.lang.Math;
/* Defines a base-door with
   no particular features */
public interface TDoor {
  /* State references */
  public boolean getLocked();
  /* Coin management */
  public int getDoorMaxCoins();
  /* Tells if the door is locked or not */
  default boolean isLocked()
    {return getLocked();}
  /* Tries to open the door */
  default int open() {
    if(!isLocked()) {
      out.println("The door has been opened!");
      double rnd = Math.random();
      int cns = (int)(rnd * getDoorMaxCoins())+1;
      out.println("You got "+ cns +" coins.");
      return cns;
    }
    else {
      out.println("This door is locked.");
      return -1;
    }
  }
  /* Performs a knock on the door */
  default int knock() {
    out.print("Door says: ");
    out.print("How you dare, ");
    out.println("I am the one who knocks!");
    int c = (Math.random()<0.8)? 0 : 1;
    if(c > 0)
      out.println("Ow! You got a free coin!");
    return c;
  }
}
```

```
import java.lang.Math;
/* Provides a counter that after
   a limit releases coins */
public interface TCounter {
  /* State references */
  public int getCounter();
  public void setCounter(int c);
  public int getLimit();
  /* Coin management */
  public int getCounterMaxCoins();
  default void incrementCounter() {
    setCounter(getCounter()+1);
  }
  default void decrementCounter() {
    setCounter(getCounter()-1);
  }
  default boolean hasReachedLimit() {
    return getCounter() >= getLimit());
  }
  default int releaseCoins() {
    double rnd = Math.random();
    int cns = (int)(rnd * getCounterMaxCoins())+1;
    out.println("You got "+cns+" coins.");
    return cns;
  }
}
```

```
import java.lang.Math;
/* Provides a chest that contains coins */
public interface TChest {
  /* Coin management */
  public int getChestMaxCoins();
  /* Opens the chest */
  default int open() {
    out.print("The chest is now opened!");
```

```
    double rnd = Math.random();
    int c = (int) (rnd * getChestMaxCoins());
    out.print("You got "+c);
    out.println(" coins from the chest.");
    return c;
  }
}

import java.lang.Math;
/* Provides an enchantment that
   can give or take coins */
public interface TEnchantment {
  /* Coin management */
  public int getEnchantMaxCoins();
  /** An enchantment can give coins
      (max getEnchantMaxCoins()) or
      remove coins (max -getEnchantMaxCoins()) **/
  default int applyEnchantment() {
    out.println("\nThis is an enchantment!");
    out.print("\"If the luck is up, ");
    out.println("of coins you'll have a cup,");
    out.print("but if no luck you got, ");
    out.println("you are gonna lose a lot.\"");
    int max = getEnchantMaxCoins();
    double rnd = Math.random();
    int cns = -max + (int)(rnd*((max*2)+1));
    if(cns >= 0) {
      out.print("Ohoh! You got "+cns);
      out.println(" coins!");
    }
    else {
      out.print("You lost "+Math.abs(cns));
      out.println(" coins!");
    }
    return cns;
  }
}

/* Puts together a door and an enchantment */
public interface TEnchantedDoor
    extends TDoor, TEnchantment {
  /** When you open an enchanted door,
    you break the enchantment and so
    you apply it.
   **/
  default int open() {
    int coins = TDoor.super.open();
    if(coins > 0) //if the door is open
      coins += applyEnchantment();
    return coins;
  }
}

public class EnchantedDoor
    implements TEnchantedDoor {
  /* Fields for the door */
  private boolean locked;
  /* Glue Code for TDoor */
  public boolean getLocked()
    {return locked;}
  /* Glue code for coin management */
  public int getDoorMaxCoins()
    {return 120;}
  public int getEnchantMaxCoins()
    {return 150;}
  /* Constructor */
  public EnchantedDoor(boolean l)
    {setLocked(l);}
  /* Other helpful methods */
  public void setLocked(boolean l)
    {this.locked = l;}
}

/* Puts together a door and a chest */
public interface TChestedDoor
    extends TDoor, TChest {
  /** When you open a chested door,
    you also get the prize from the chest.
    This overrides the TDoor's open().
   **/
  default int open() {
  int coins = TDoor.super.open();
  if(coins > 0) //if the door is open
    coins += openChest();
  return coins;
  }
  /* Alias for open() from TChest */
  default int openChest()
    {return TChest.super.open();}
}

public class ChestedDoor
    implements TChestedDoor {
  /* Fields for the door */
  private boolean locked;
  /* Glue Code for TDoor */
  public boolean getLocked()
    {return this.locked;}
  /* Glue code for coin management */
  public int getDoorMaxCoins()
    {return 120;}
  public int getChestMaxCoins()
    {return 250;}
  /* Constructor */
  public ChestedDoor(boolean l)
    {setLocked(l);}
  /* Other helpful methods */
  public void setLocked(boolean l)
    {this.locked = l;}
}

/* Puts together a door and a counter */
public interface TKnockDoor
    extends TDoor, TCounter {
  /** Every knock makes the counter increment.
    If the limit is reached, more coins are
    released. **/
  default int knock() {
    int coins = TDoor.super.knock();
    incrementCounter();
    if(hasReachedLimit()) {
      out.print("Ohh! A special drop for you!");
      coins += releaseCoins();
    }
    else {
      //Let's give a suggestion to the player
      out.print("Don't challenge me... ");
      int c = getLimit();
      String sug = "never knock a door ";
      sug = sug + "more then "+c+" times.";
      out.println(sug);
    }
    return coins;
  }
}

public class KnockDoor implements TKnockDoor {
  /* Fields for the door */
  private boolean locked;
  /* Fields for the counter */
  private int counter;
  private int limit;
  /* Glue Code for TDoor */
  public boolean getLocked()
    {return this.locked;}
```

```
  /* Glue code for TCounter */
  public int getCounter()
    {return this.counter;}
  public void setCounter(int c)
    {this.counter = c;}
  public int getLimit()
    {return this.limit;}
  /* Glue code for coin management */
  public int getDoorMaxCoins()
    {return 120;}
  public int getCounterMaxCoins()
    {return 500;}
  /* Constructor */
  public KnockDoor (boolean l, int li) {
    setCounter(0);
    setLocked(l);
    setLimit(li);
  }
  /* Other helpful methods */
  private void setLocked(boolean l)
    {this.locked = l;}
  private void setLimit(int l)
    {this.limit = l;}
}

public class DoorsRoom {
  private TDoor leftDoor;
  private TDoor rightDoor;
  private TDoor frontDoor;
  /* Constructor */
  public DoorsRoom(TDoor l,
                   TDoor r,
                   TDoor f) {
    leftDoor = l;
    rightDoor = r;
    frontDoor = f;
  }
  /* Getters */
  public TDoor getLeftDoor()
    {return leftDoor;}
  public TDoor getRightDoor()
    {return rightDoor;}
  public TDoor getFrontDoor()
    {return frontDoor;}
}

public class Player {
  private final String nickname;
  private int bag = 150; //contains the coins
  /* Constructor */
  public Player(String n) {
    nickname = n;
  }
  /* Setters and getters */
  public int getCoins()
    {return bag;}
  public String getNickname()
    {return this.nickname;}
  /* Helpful methods */
  public void addInBag(int amount)
    {this.bag += amount;}
  public void removeFromBag(int amount)
    {this.bag -= amount;}
  public String toString() {
    String s = "I'm "+getNickname();
    s += " and i've got "+getCoins();
    s += " coins in my bag.";
    return s;
  }
}

public class Game {
  private Player player;
  private DoorsRoom doorsRoom;
  private final String version = "0.0";
  /* Constructor */
  public Game(Player p, DoorsRoom dr) {
    player = p;
    doorsRoom = dr;
  }
  /* Setters and getters */
  public Player getPlayer()
    {return player;}
  public DoorsRoom getDoorsRoom()
    {return doorsRoom;}
}
```